To create the hash table for the given single-character strings, we need to compute the hash code for each character, then take the modulus with 5 to determine the index in the hash table.

In Java, `String.hashCode()` is computed using the formula:

$$\text{hash} = s[0] \times 31^{(n-1)} + s[1] \times 31^{(n-2)} + \ldots + s[n-1]$$

For single-character strings, the hash code is simply the Unicode value of the character. Therefore, the hash code for each character is its ASCII value.To compute the hash values for the single-character strings "E", "A", "S", "Y", "Q", "U", "E", "S", "T", "I", "O", "N" and then draw the hash table, we need to follow these steps:

1. Calculate the hash code for each character.
2. Compute the modulo 5 of each hash code.
3. Insert each character into the hash table at the position given by the modulo operation, associating each character with its index.:
4. **E**: ASCII value = 69, `69 % 5 = 4`
5. **A**: ASCII value = 65, `65 % 5 = 0`
6. **S**: ASCII value = 83, `83 % 5 = 3`
7. **Y**: ASCII value = 89, `89 % 5 = 4`
8. **Q**: ASCII value = 81, `81 % 5 = 1`
9. **U**: ASCII value = 85, `85 % 5 = 0`
10. **E**: ASCII value = 69, `69 % 5 = 4`
11. **S**: ASCII value = 83, `83 % 5 = 3`
12. **T**: ASCII value = 84, `84 % 5 = 4`
13. **I**: ASCII value = 73, `73 % 5 = 3`
14. **O**: ASCII value = 79, `79 % 5 = 4`
15. **N**: ASCII value = 78, `78 % 5 = 3`

Now let's construct the hash table. Each entry will list the index in the table and the value (i) associated with each character. Since we're inserting keys sequentially, we'll note collisions and insert values using a simple linear probing method:

```
Index 0: [A: 1, U: 5]
Index 1: [Q: 4]
Index 2: []
Index 3: [S: 2, S: 7, I: 9, N: 11]
Index 4: [E: 0, Y: 3, E: 6, T: 8, O: 10]
```

Explanation of Linear Probing:

1. **E** at index 4.
2. **A** at index 0.
3. **S** at index 3.
4. **Y** at index 4. (collision with E, use next available: index 4)
5. **Q** at index 1.
6. **U** at index 0. (collision with A, use next available: index 1)
7. **E** at index 4. (collision with E and Y, use next available: index 4)
8. **S** at index 3. (collision with S, use next available: index 3)
9. **T** at index 4. (collision with E, Y, E, use next available: index 4)
10. **I** at index 3. (collision with S, S, use next available: index 3)
11. **O** at index 4. (collision with E, Y, E, T, use next available: index 4)
12. **N** at index 3. (collision with S, S, I, use next available: index 3)

Each entry in the hash table includes the character and its associated index (i) from the sequence. Linear probing ensures that even if a collision occurs, the value can still be inserted by checking the next available spot in the table.

In the context of hash tables, a **collision** occurs when two different keys hash to the same index or slot in the hash table. Since a hash table is typically implemented as an array, each key-value pair is placed at an index determined by the hash of the key. When two keys produce the same hash value, they both attempt to occupy the same index, leading to a collision.

Handling Collisions

There are several strategies for handling collisions:

1. **Chaining:** This approach involves maintaining a list (or another data structure, like a linked list) at each index of the hash table. When a collision occurs, the new key-value pair is added to the list at the hashed index.

    - **Example:** If both keys "cat" and "dog" hash to index 3, they are both stored in a list at index 3.

    ```
    Index 3: ["cat": value1, "dog": value2]
    ```
2. **Open Addressing:** This strategy involves finding another available spot in the hash table when a collision occurs. There are several methods of open addressing:

- **Linear Probing:** Sequentially check the next slots in the array until an empty one is found.
    - **Example:** If index 3 is occupied, try index 4, then index 5, etc.
- **Quadratic Probing:** Similar to linear probing but uses a quadratic function to determine the next slot.
    - **Example:** If index 3 is occupied, try index 4, then index 7 (3 + 2^2), then index 12 (3 + 3^2), etc.
- **Double Hashing:** Uses a second hash function to determine the next slot.
    - **Example:** If the first hash function results in a collision at index 3, the second hash function determines the next slot to check.

3. **Rehashing:** When the hash table becomes too full or collisions occur too frequently, the table can be resized, and all the keys can be rehashed into a larger table to reduce collisions.

Handling collisions effectively is crucial for maintaining the performance of a hash table, as excessive collisions can degrade the average time complexity for operations such as insertions, deletions, and lookups from $O(1)$ to $O(n)$.

Hash Table Construction with chaining

Given these results, we will insert each character into the hash table at the computed index, associating each character with its index in the sequence (0 to 11). If a collision occurs (i.e., two characters hash to the same index), we will use chaining to handle it.

Here is the hash table:

Index	Values
0	A (1), U (5)
1	Q (4)
2	
3	S (2), S (7), I (9), N (11)
4	E (0), Y (3), E (6), T (8), O (10)

Explanation

- **Index 0**: 'A' (index 1) and 'U' (index 5) both hash to 0.
- **Index 1**: 'Q' (index 4) hashes to 1.
- **Index 2**: No characters hash to 2.
- **Index 3**: 'S' (index 2), 'S' (index 7), 'I' (index 9), and 'N' (index 11) all hash to 3.
- **Index 4**: 'E' (index 0), 'Y' (index 3), 'E' (index 6), 'T' (index 8), and 'O' (index 10) all hash to 4.

This hash table uses chaining to handle collisions, where each index points to a list of values that hash to that index.